Project 1

Aahlad Madireddy
Alejandro Arredondo

Exercise 1

n/a – ex1.c is included in project directory.

Exercise 2

n/a

Exercise 3

Identify the exact assembly instructions that correspond to each of the statements in readsect()

readsect() source	**Corresponding asm**
`waitdisk()`	`0x7cf4: call 0x7cda`

`outb(0x1F2, 1);`	`0x7cf9: mov $0x1f2,%edx`
	`0x7cfe: mov $0x1,%al`
	`0x7d00: out %al,(%dx)`

`outb(0x1F3, offset);`	`0x7d01: movzbl %bl,%eax`
	`0x7d04: mov $0xf3,%dl`
	`0x7d06: out %al,(%dx)`

`outb(0x1F4, offset >> 8);`	`0x7d07: movzbl %bh,%eax`
	`0x7d0a: mov $0xf4,%dl`
	`0x7d0c: out %al,(%dx)`

`outb(0x1F5, offset >> 16);`	`0x7d0d: mov %ebx,%eax`
	`0x7d0f: mov $0xf5,%dl`
	`0x7d11: shr $0x10,%eax`
	`0x7d14: movzbl %al,%eax`
	`0x7d17: out %al,(%dx)`

`outb(0x1F6, (offset >> 24) | 0xE0);`	`0x7d18: shr $0x18,%ebx`
	`0x7d1b: mov $0xf6,%dl`
	`0x7d1d: mov %bl,%al`
	`0x7d1f: or $0xffffffe0,%eax`
	`0x7d22: out %al,(%dx)`

`outb(0x1F7, 0x20);`	`0x7d23: mov $0x20,%al`
	`0x7d25: mov $0xf7,%dl`
	`0x7d27: out %al,(%dx)`

`waitdisk();`	`0x7d28: call 0x7cda`

`insl(0x1F0, dst, SECTSIZE/4);`	`0x7d2d: mov 0x8(%ebp),%edi`
	`0x7d30: mov $0x80,%ecx`
	`0x7d35: mov $0x1f0,%edx`
	`0x7d3a: cld`
	`0x7d3b: repnz insl (%dx),%es:(%edi)`

Identify the begin and end of the for loop that reads the remaining sectors of the kernel from the disk

Beginning: 0x7db6, end: 0x7dce

```
0x7db6:  cmp    %esi,%ebx
0x7db8:  jae    0x7dd0
0x7dba:  pushl  0x8(%ebx)
0x7dbd:  add    $0x38,%ebx
0x7dc0:  pushl  -0x10(%ebx)
0x7dc3:  pushl  -0x20(%ebx)
0x7dc6:  call   0x7d41
0x7dcb:  add    $0xc,%esp
0x7dce:  jmp    0x7db6
```

Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint

1. **At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?**
    a. Right before the call into bootmain, it switches to 32 bit. Setting the protected mode flag and then doing a long jump causes the switch.

2. **What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?**
    a. Last (two) instructions of bootloader:
    ```
    0x7dd5:  mov    %eax,%ebx
    0x7dd7:  call   *0x10018
    ```
    b. First of kernel:
    ```
    movl $multiboot_info, %eax
    ```

3. **How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?**
    a. In line 54 of boot/main.c, it checks whether the program has written to 0x10000, and loads in a segment when it hasn't. It gets all the information from the ELF header.

Exercise 5

Identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong.

1. The first thing to break will be the transition into protected mode, because this depends on the link address being right for the GDT segment translation. The first instruction that won't work is `ljmp $PROT_MODE_CSEG, $protcseg`

Exercise 6

Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)

Before the boot loader:

```
(gdb) x/8x 0x0100000
0x100000 <_head64>:     0x00000000  0x00000000  0x00000000  0x00000000
0x100010 <_head64+16>:  0x00000000  0x00000000  0x00000000  0x00000000
```

After the boot loader:

```
(gdb) x/8x 0x0100000
0x100000 <_head64>:     0x107000b8  0x66188900  0x047205c7  0x12340000
0x100010 <_head64+16>:  0x007c00bc  0x00cce800  0x20b80000  0x0f000000
```

When the BIOS enters the boot loader the kernel hasn't been loaded into memory so the address 0x010000 is empty. After the boot loader has loaded the kernel the address updates accordingly. The second breakpoint after 0x7c00 shows us the point at which the kernel is loaded into memory.

Exercise 7

What is the first instruction after the new mapping is established that would fail to work properly if the old mapping were still in place? Comment out or otherwise intentionally break the segmentation setup code in kern/entry.S , trace into it, and see if you were right.

The file obj/kern/kernel.asm outlines everything occurring when the bootstrap initializes the mapping of the first 1G of memory at KERNBASE

Starting at line 93(0x00100074):

```
100072:     89 1f       mov   %ebx,(%rdi)
```

```
        # setting the pgdir so that the LA=PA
        # mapping first 1G of mem at KERNBASE
        movl $128,%ecx

100074:     b9 80 00 00 00    mov   $0x80,%ecx
        # Start at the end and work backwards
        #leal (pml4 + 5*0x1000 - 0x8),%edi
        movl  $pde1,%edi
```

The page is loaded into cr3 at line 131(0x001000aa):

```
1000a3:     75 e9       jne   10008e <_head64+0x8e>
        /*    subl $1,%ecx */
        /*    cmp $0x0,%ecx */
        /*    jne 1b */
        # set the cr3 register
        movl $pml4,%eax

1000a5:     b8 00 20 10 00    mov   $0x102000,%eax
        movl  %eax, %cr3

1000aa:     0f 22 d8    mov   %rax,%cr3
        # enable the long mode in MSR
        movl $EFER_MSR,%ecx
```

However, if we comment out line 105(0x00100083):

```
100083:     81 c3 00 01 00 00       add   $0x100,%ebx
        # PTE_P|PTE_W|PTE_MBZ
        movl $0x00000183,%eax
```

We'll see that the bootloader can not continue in long mode because we have prevented register %eax from being updated with the paging info.

Exercise 9

Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

The kernel's stack is located at 0x8004000000. The kernel reserves space by clearing the frame pointer by setting it to 0 and then setting the stack pointer, before it calls the i386init function. The "top" of this reserved area is where the stack pointer starts, and the stack grows downwards.

Exercise 10

To become familiar with the C calling conventions on the x86-64, find the address of the test_backtrace function in obj/kern/kernel.asm, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 64-bit words does each recursive nesting level of test_backtrace push on the stack, and what are those words?

test_backtrace 5:

```
#0  test_backtrace (x=128) at kern/init.c:21
#1  0x000000800420009d in test_backtrace (x=5) at kern/init.c:24
#2  0x0000008004200185 in i386_init () at kern/init.c:55
#3  0x0000008004200056 in relocated () at kern/entry.S:76
#4  0x0000000000000000 in ?? ()
```

test_backtrace 4:

```
#0  test_backtrace (x=128) at kern/init.c:21
#1  0x000000800420009d in test_backtrace (x=4) at kern/init.c:24
#2  0x000000800420009d in test_backtrace (x=5) at kern/init.c:24
#3  0x0000008004200185 in i386_init () at kern/init.c:55
#4  0x0000008004200056 in relocated () at kern/entry.S:76
#5  0x0000000000000000 in ?? ()
```

test_backtrace 3:

```
#0  test_backtrace (x=0) at kern/init.c:21
#1  0x000000800420009d in test_backtrace (x=3) at kern/init.c:24
#2  0x000000800420009d in test_backtrace (x=4) at kern/init.c:24
#3  0x000000800420009d in test_backtrace (x=5) at kern/init.c:24
#4  0x0000008004200185 in i386_init () at kern/init.c:55
#5  0x0000008004200056 in relocated () at kern/entry.S:76
#6  0x0000000000000000 in ?? ()
```

test_backtrace 2:

```
#0  test_backtrace (x=0) at kern/init.c:21
#1  0x000000800420009d in test_backtrace (x=2) at kern/init.c:24
#2  0x000000800420009d in test_backtrace (x=3) at kern/init.c:24
#3  0x000000800420009d in test_backtrace (x=4) at kern/init.c:24
#4  0x000000800420009d in test_backtrace (x=5) at kern/init.c:24
#5  0x0000008004200185 in i386_init () at kern/init.c:55
#6  0x0000008004200056 in relocated () at kern/entry.S:76
#7  0x0000000000000000 in ?? ()
```

test_backtrace 1:

```
#0  test_backtrace (x=0) at kern/init.c:21
```

```
#1  0x000000800420009d in test_backtrace (x=1) at kern/init.c:24
#2  0x000000800420009d in test_backtrace (x=2) at kern/init.c:24
#3  0x000000800420009d in test_backtrace (x=3) at kern/init.c:24
#4  0x000000800420009d in test_backtrace (x=4) at kern/init.c:24
#5  0x000000800420009d in test_backtrace (x=5) at kern/init.c:24
#6  0x0000008004200185 in i386_init () at kern/init.c:55
#7  0x0000008004200056 in relocated () at kern/entry.S:76
#8  0x0000000000000000 in ?? ()
```